Integrating Productivity-Oriented Programming Languages with High-Performance Data Structures

Rohit Varkey Thankachan[1], Eric R. Hein[2], Brian P. Swenson[3], and James P. Fairbanks[3]

***Abstract*— This paper shows that Julia provides sufficient performance to bridge the performance gap between productivity-oriented languages and low-level languages for complex memory intensive computation tasks such as graph traversal. We provide performance guidelines for using complex low-level data structures in high productivity languages and present the first parallel integration on the productivity-oriented language side for graph analysis. Performance on the Graph500 benchmark demonstrates that the Julia implementation is competitive with the native C/OpenMP implementation.**

I. INTRODUCTION

The applicability of High Performance Computing technology is limited in applications outside of the HPC research communities. A large set of existing libraries utilize complex data structures in low-level languages for specialized tasks to obtain high performance. Using high productivity languages bridges this gap between high performance computing and widespread adoption. While many researchers have integrated scripting languages with HPC codes to achieve floating point intensity [1], [2], [3], [4], there is less work on using complex, non-contiguous, low-level data structures. In contrast to codes which perform bulk operations on simple data structures, tasks such as graph traversal and network analysis require tightly coupled interactions between application-specific subroutines and data structures [5].

NetworkX (Python) [6] and LightGraphs.jl (Julia) [7] are graph analysis libraries purely implemented in productivity-oriented languages. They use simple data structures created natively in productivity-oriented languages - NetworkX uses a dictionary of dictionaries, LightGraphs uses a vector of vectors - to represent graphs. While libraries like these are convenient for small graphs, they suffer from poor scalability, especially when dealing with large dynamic graphs. For example, when dynamically building a graph out of streams of SCinet NetFlow traffic [8], researchers chose STINGER [9], [10], a complex data structure implemented in C. STINGER is a multicore parallel graph engine, using a vector-of-blocked-lists along with carefully tuned primitives for efficient multi-threaded graph construction and traversal. HPC data structures must be implemented in low-level programming languages in order to obtain good performance for large streaming graphs.

This work was supported by the Georgia Tech Research Institute

[1]School of Computer Science, Georgia Institute of Technology Atlanta, GA, USA rohitvarkey@gatech.edu

[2]School of Electrical and Computer Engineering, Georgia Institute of Technology Atlanta, GA, USA ehein6@gatech.edu

[3]Georgia Tech Research Institute, Atlanta, GA, USA brian.swenson,james.fairbanks@gtri.gatech.edu

In order to bridge the gap between rapid development and high performance, software for graph analysis has moved towards a hybrid model of using a high-productivity language as an interface for users to a high-performance language which implements the performance critical code for algorithms and low-level complex data structures. The Stanford Network Analysis Platform (SNAP)[11], [12] is a high performance network analysis package that supports productivity-oriented language integration through a Python interface to a C++ core. The network analysis package igraph[13] uses C for performance critical code, exposing interfaces to Python and R. Similarly, graph-tool[14] is a Python package built on the Boost Graph library[15] for C++. NetworKit[16] follows the same philosophy and uses a Python interface to a C++ core. The SEJITS[5] project allows for the definition of embedded Domain Specific Languages (DSL) for selective embedded just in time (JIT) compilation to obtain high performance for higher level languages. The Knowledge Discovery Toolbox (KDT)[17], [18] provides a Python interface for analysis of large semantic graphs using filters to define graph operations and SEJITS to provide high performance. KDT achieves high performance by using an embedded DSL to compile selective parts of productivity-oriented languages to lower level languages. Python bindings to STINGER were developed using SWIG[19] to provide a productivity-oriented programming language interface to STINGER. Support for these bindings was discontinued due to the lack of a multi-threaded graph traversal capability. These efforts underline the necessity of higher level languages in graph analysis. However, they suffer from the two language problem[20] of requiring a productivity-oriented language for rapid iteration and development and another low-level language to achieve good performance. Our effort focuses on implementing graph algorithms completely in a productivity-oriented language while reusing only the low-level complex data structure from C, which is necessary to truly solve the two language problem.

Graph analysis must be parallelized to fully utilize modern computing resources. STINGER, SNAP, graph-tool, and NetworKit use shared memory OpenMP parallelism in their core implementations to parallelize graph algorithms while igraph does not offer parallelism. KDT offers parallelism through its C++/MPI backend. Unlike Python [21] and Ruby, threading primitives in Julia achieve efficient parallelism because there is no Global Interpreter Lock (GIL) restricting parallelism by serializing interpreter operations. The GIL makes parallelizing NetworkX and the Python interfaces to complex low level data structures infeasible. Our package is able to

use threading constructs in the high productivity language (Julia) rather than only in low-level language constructs, allowing graph algorithms to be more easily parallelized without sacrificing performance.

We make the following contributions:

1) First integration with a complex HPC graph data structure that allows efficient parallel algorithms to be expressed natively in a productivity-oriented language
2) First rigorous study of Julia atomics and low-level integration performance characteristics.
3) New atomic collections for Julia that avoid performance pitfalls.

II. Methodology

In order to study the problem of producing high-performance, high-productivity tools for graph analysis, we undertook the task of making the STINGER data structure accessible within the Julia language. In the following sections we provide an introduction to the STINGER data structure and the Julia integration interfaces, discuss how we mitigated several performance pitfalls and complicating factors, and provide empirical data justifying our design choices.

A. The STINGER graph data structure

We use the STINGER[9], [10] data structure in our experiments. STINGER is a graph data structure that supports rapid insertions and deletions, especially when compared to other open-source graph database software packages[22]. The implementation uses many features of low-level C code, including flexible array members, manual memory allocation, custom synchronization primitives, and atomic memory operations.

Several aspects of the STINGER design enable high performance streaming graph analytics. STINGER mitigates the overhead of dynamic memory allocations by preallocating one large chunk of memory at startup. Several offsets are stored to carve all the necessary sub-structures from this chunk. This design allows multiple processes to map the data structure into shared memory without invalidating internal pointers. The STINGER graph data structure consists of a contiguous array for vertex storage and a pool of edge blocks for edge storage. An edge block is a small contiguous array of edges. Each edge within the block is initially empty, but once filled will store a weight, the time of its creation, and the time it was last updated in addition to the destination vertex. The adjacency list of each vertex is stored as a linked-list of edge blocks. This design conveys some of the benefits of cache-friendly contiguous data structures while still allowing for efficient modification. Multiple threads can read and modify an adjacency list simultaneously. Atomic memory operations are used to synchronously modify edge data and append edge blocks to the list.

B. Integrating Julia with STINGER

Julia provides an elegant foreign function interface (FFI) [23] for interacting with C and Fortran libraries. The Julia FFI provides a mapping of Julia data structures to C structures which can be used to create Julia data structures that are analogous to C data structures. These Julia data structures can be passed to any C function that takes its C counterpart as an argument using the `ccall` function. This approach is ideal as it allows for the memory allocations to be done inside Julia.

However, the Julia C FFI structure mappings do not support mapping the flexible array members in the STINGER data structure. A flexible array member is declared as an array with zero length at the end of a struct. When an instance of the struct is allocated, the size of this member can be controlled by adding bytes to the total size of the memory allocation. Understanding the layout of the STINGER data structure requires both the static struct definitions and the run-time array size information. Modification of C structs to comply with the constraints imposed by Julia is not always feasible for established libraries that are currently in use like STINGER.

The memory for STINGER, or any C data structure that cannot be completely mapped by Julia, must be allocated in C. A handle to the C allocated memory must be maintained to access this memory from Julia. These restrictions are met by creating a Julia type that stores the pointer to the C data structure as a handle. The initialization function that allocates the memory for STINGER in C is called from Julia on the creation of this Julia type. Memory is freed by calling the C deinitialization function in the finalizer of the Julia type which is called when the Julia garbage collector frees the Julia object. This approach provides the user with Julia memory management semantics when working with the Julia type by allowing the Julia garbage collector (GC) to control allocations and deallocations in the C heap. However, the Julia GC only knows the size of the pointer to the object and does not know the size of the object in the C heap itself. Therefore, the GC does not prioritize freeing the C object when it feels memory pressure. This is another reason why it is preferable to use the complete mapping between Julia and C if possible.

C. Interacting with C struct fields

The dual memory model described in Section II-B has a heap managed by Julia, and a heap managed by STINGER through calls to `malloc`. The two memory spaces are separate and not synchronized. To interact with the fields of the C structure in Julia, the Julia heap has to load/store from/to the C heap using the handle of the C struct for each load/store to these fields. A Julia representation of the fields in the non-mappable C struct has to be maintained to interact with these fields using Julia syntax. Both a lazy or eager approach to maintaining such a representation can be followed. Care must be taken when devising strategies for synchronizing data structures between the Julia and C memory spaces.

Using the eager approach, even though we can decode the C structure using a pointer to a type in Julia, this does not update when the C memory updates. This puts the onus on

the Julia code to maintain consistency with the C memory. This could be done by making sure to load the representation every time a `ccall` occurs. When using a lazy approach, we have the additional overhead of having to load the memory when the user is trying to access it, rather than having an already loaded object in Julia memory that can be accessed. Table I summarizes the actions on each operation for both approaches.

TABLE I
LAZY VS EAGER

Operation	Eager	Lazy
getfields	Already cached	Load pointer
setfields	Store pointer	Store pointer
ccalls	Loads occur for every `ccall`	No op

The time taken can be modeled as

$$T = \alpha S + \beta L + \gamma C$$

where S is the number of sets, L is the number of loads and C is the number of ccalls and α, β, γ are machine specific constants in units of $s/operation$.

For the eager approach,

$$T_e = \alpha SF + \beta C + \gamma C$$

where SF is the number of setfields and C is the number of C calls.

For the lazy approach,

$$T_l = \alpha SF + \beta GF + \gamma C$$

where SF is the number of setfields, GF is the number of getfields and C is the number of C calls.

$$T_e - T_l = \beta(C - GF) \tag{1}$$

In general we expect workflows to have many more getfields than ccalls ($GF > C$). In these cases equation 1 recommends the lazy approach to synchronization as the most efficient choice. Until established codes can be ported or support for flexible array members is provided by the FFI, Julia codes that map complex C data structures will need to take this approach.

D. Walking data structures directly

Besides mapping the memory for STINGER, the Julia code must also understand the data layout in order to traverse the successors of a vertex, such as in the BFS algorithm. The STINGER data structure stores data representing the edges of the graph as edge blocks in the flexible array member. One option to read these edges is to use functions written in STINGER that read the edge blocks and return an array of successors to Julia. However, this approach requires gathering the neighbors from the edge blocks into a contiguous array which is costly and hurts performance significantly when used in a tight loop. STINGER implements iteration over edges as a C macro to avoid this slow gathering step as much as possible.

The second option is to walk the data structure directly from Julia to read the edge blocks similar to the C macro used in STINGER. The pointer offsets stored in the metadata fields of the STINGER data structure are used to correctly compute the memory address to the required edge blocks and then load the edges into Julia. A function passed in as an argument to the iterator function is executed on each edge loaded in. This option does not incur the cost involved with gathering the successors into an array.

A pair of traps can be identified here:

1) Julia pointer arithmetic semantics differ from C. Julia does not increment in multiples of the size of the object but rather as just bytes. Hence, while computing the memory location, care needs to be taken to take the product of the offset and the size of the object the pointer points to.
2) Structure padding performed by compilers might result in the sizes of objects being different than the expected size. Explicit padding done on the C side for most STINGER data structures helped avoid this ambiguity in several cases.

Like C++, Julia supports iterators, which encapsulate the logic required to move between consecutive elements of a data structure. Using this technique one can load only required memory from C rather than copying the entire structure at the outset, thus reducing the memory bandwidth required. The benefit of iterating directly over the edge blocks versus gathering successors into a contiguous array before processing must be determined experimentally. In this microbenchmark we iterate through the first 1000 vertices of a Kronecker graph in random order, touching all the edges for each vertex. We also benchmarked both approaches using the BFS benchmark. Table II shows the results. While the iterators are twice as fast as the gathering successors approach in the experiment, in the BFS implementation the iterators are around $6x$ faster. This can be attributed to these performance differences getting magnified in a real world workload. We recommend directly walking low-level complex data structures in Julia in performance critical code.

TABLE II
ITERATORS (I) VS GATHERING SUCCESSORS (G) – ALL TIMES IN MS

Scale	Exp (I)	Exp (G)	BFS (I)	BFS (G)
10	1.03	2.43	252.17	1833.70
11	2.21	4.92	504.37	3623.40
12	4.64	10.33	1034.36	7239.56
13	9.70	21.04	2142.28	14461.98
14	20.79	44.18	4328.72	28767.98
15	58.11	107.91	12583.00	67962.16
16	127.92	225.55	27036.85	128637.68

E. Parallelism in Julia

Julia has support for several models of parallelism available.

1) Remote process execution for a distributed environment through Channels with Remote Calls and Remote References similar to the MPI[24] parallelism model.

2) Lightweight "green" threading through Tasks similar to the parallelism model in Cilk[25].
3) Native multi-threading support similar to the OpenMP[26] parallelism model.

Multi-threading is a recently added model to Julia and is an experimental feature in Julia 0.5. It exposes a `@threads` macro which can be used to parallelize loops by affixing it in front of a for loop, similar to `#pragma OMP parallel for`. The `@threads` loop statically partitions the work based on loop index without any work stealing.

We use the multi-threading model to parallelize the BFS implementation in our experiments. This model suits STINGER as the data structure is stored on a single machine and multi-threading allows the use of multiple cores with a single copy of the data structure in shared memory.

Julia is a dynamically typed programming language that relies on the ability to compile specific methods based on multiple dispatch to generate fast code using its JIT compiler. It performs type inference to infer the type of a variable from the code at compile time. It is a general recommendation to obtain good performance for any Julia code to use type stable code. Type stable functions are functions for which the JIT compiler is able to infer the types of the variables and the return type at compile time. Multi-threaded Julia code has issues with type inference and the compiler does not infer the return type of a function correctly at times. This can lead to disastrous performance results as functions with type instabilities are much slower than type stable functions. This inference issue causes the compiler to unnecessarily box some variables even when it is not required, leading to a lot more allocations and loss in performance. We highly recommend using the `@code_warntype` macro to ensure that the compiler recognizes the loop body functions as type stable. Using function barriers to separate the kernel, i.e, splitting out the `@threads` for loop into another function can help remove unnecessary boxing of variables. This is in contrast to languages like C and Fortran where manual inlining is a common technique for improving performance. It is possible to write fast multi-threaded code easily in Julia using the `@threads` parallelism model by taking care to ensure type stability.

F. Atomics in Julia

Good support for atomic operations is an essential part of any programming language that supports parallelism. Julia provides a module in Base to support atomic operations on its Integer and Floating point types using LLVM's atomic intrinsics. Julia requires the developer to create an `Atomic` type using the value, which stores a reference to the value. This type ensures that only atomic operations are performed on these types. Runtime exceptions are raised to prevent non-atomic operations.

A thread safe queue data structure created using these atomic types is used for the multi-threaded BFS implementation. The head and the tail of the queue are Atomic Integers that are accessed and modified atomically. A standard Vector stores the values in the queue. Native Julia atomic support is sufficient for a high-performance implementation of this data structure.

However, the native Julia atomic support is insufficient for all of our needs. For example, we required a vector of integers that needed to be atomically accessed in our BFS implementation. A vector of native Julia atomic integers is actually a vector of references to the values. This adds a layer of indirection to each access which hurts performance. A global lock on the vector to ensure only one thread can access the entire vector at a time would guarantee correctness but reduces concurrency.

Our workload requires new atomic primitives in the Julia environment as both existing solutions were inefficient. We created UnsafeAtomics.jl, a package that adds support for atomic operations on normal Vectors by bypassing the usual safety requirement that the contents to be instances of the Julia Atomic type. In idiomatic Julia code, the prefix "unsafe" refers to operations without correctness guarantees such as dereferencing arbitrary pointers, which can cause memory segmentation faults. These unsafe operations are necessary for efficient interoperability with many C libraries, which use pointer arithmetic. The UnsafeAtomics approach removes a layer of indirection, allowing for faster execution of LLVM atomic instructions. Table II-F presents a microbenchmark comparing the performance of the two atomic approaches, in which threads repeatedly execute an atomic compare-and-swap operation on randomly-chosen elements of a Vector. The chosen array size yields low contention and an irregular accesses pattern. The UnsafeAtomics reduce the run time of the Graph BFS benchmark with 64 randomly chosen vertices. The new atomic primitives are 20-30% faster in the microbenchmark and around 20% faster in the BFS benchmark. As the size of the graph increases, the relative performance difference between these approaches increases. These difference vary based on machine-specific factors, but in general, removing the extra layer of indirection is critical to achieving parallel speedup in high level languages.

Scale	Exp (N)	Exp (U)	Exp(N)/ Exp(U)	BFS (N)	BFS (U)	BFS(N)/ BFS(U)
10	0.13	0.1	1.3	47.23	43.27	1.10
11	0.27	0.23	1.17	98.99	91.32	1.08
12	0.62	0.47	1.32	217.44	190.74	1.14
13	1.31	0.97	1.35	505.59	420.84	1.20
14	2.7	2.17	1.24	1158.3	977.1	1.185
15	5.74	3.93	1.46	2576.18	2154.5	1.20
16	11.6	8.77	1.32	5565.87	4559.16	1.22

TABLE III
ATOMICS: NATIVE (N) VS UNSAFE (U) (TIMES IN MS)

Julia makes it feasible for developers to extend the language and add support for features required by the developer. These vectors of atomics will allow for the construction of concurrent data structures and are built by emitting the appropriate LLVM intrinsics directly. No other widely used productivity-oriented language allows such seamless use of compiler intrinsic instructions.

```
function bfskernel(
  alg::LevelSynchronous, s::Stinger,
  next::ThreadQueue,
  parents::Array{Int64},
  level::Array{Int64}
 )
 @threads for src in level
  foralledges(s, src) do edge, src, etype
   direction, neighbor = edgeparse(edge)
   if (direction != 1)
    parent = UnsafeAtomics.unsafe_atomic_cas!(
     parents, neighbor+1, -2, src
    )
    if parent==-2
     #Push onto queue
     push!(next, neighbor)
    end
   end
  end
end

function bfs(
    alg::LevelSynchronous, s::Stinger,
    next::ThreadQueue, source::Int64,
    parents::Array{Int64}
  )
 #Set source to -1
 parents[source+1]=-1
 push!(next, source)
 while !isempty(next)
  level=next[next.head[]:next.tail[]-1]
  #reset the queue
  next.head[] = next.tail[]
  bfskernel(alg,s,next,parents,level)
 end
 return parents
end
```

Listing 1: Julia multithreaded BFS code is both simple and efficient.

G. The Stinger.jl BFS implementation

We use a straightforward implementation of BFS in Julia. The operations are conducted in Julia code instead of using `ccalls`. The only point of interaction with the C data structure is accessing the neighbors of a particular vertex. We use the iterator described in Section II-D to obtain the neighbors of a given vertex. This allows us to measure the performance of using a C data structure from Julia rather than algorithms implemented in C.

This is in contrast to Numpy/Scipy which uses algorithms written mostly in C with small interfaces in Python. The path to efficiency in scientific Python is to push as much of the code into the C side as possible. We chose Julia because it is possible to write most of the code in Julia and use only primitive data structure manipulations in C.

A frontier based parallelized version[27] of BFS was implemented by using the multi-threading parallel model. We use a thread-safe queue and the `@threads` macro to parallelize the loop that explores the vertices of each frontier. The new atomic primitives defined in UnsafeAtomics.jl allowed for updating the parents array concurrently without requiring a layer of indirection. The code for the parallel BFS implementation is attached in Listing 1. The Julia parallel BFS code looks like pseudo-code because of the ability to use efficient abstractions. This demonstrates how algorithms can be written in Julia while only needing to use the C data structure to obtain data from it and how parallelism is easy to achieve in Julia.

H. The C BFS implementation

The existing STINGER C BFS implementation is also a frontier based parallel implementation of BFS using OpenMP for the parallelism. It marks vertices seen in the BFS atomically and adds them to the queue if they are seen for the first time. The BFS implementation in STINGER distributed by the Dynograph[28] benchmark suite is used in our experiments. This implementation and benchmark is used as the representative for an implementation completely in a low-level language.

I. Benchmark

The Graph500 benchmark[29] is a standard benchmark used for graphs. A Kronecker graph[30] is created and loaded into the graph analysis system and breadth-first searches are performed on randomly chosen vertices in this benchmark. We wrote a Kronecker graph generator in Julia based on the reference implementation[1] that generates a graph with 2^{scale} vertices and an edge factor of 16, varying the scale from 10 to 27.

We randomly choose 64 BFS source vertices that are connected to at least one other vertex as specified by the Graph500 specification. Both implementations use the same list of randomly chosen vertices. Our benchmark only takes into account the time taken to run the BFS and does not include the time taken for setting up the graph. BenchmarkTools.jl[31] was used for benchmarking the Julia BFS implementations. We ran the BFS benchmarks with 1, 6, 12, 24, and 48 threads for Stinger.jl and STINGER at all scales. The benchmark results can be reproduced by running the StingerWrapper-Benchmarks[32] repository. The benchmarks were run on a dual-socket server system with Intel Xeon E5-2687W v4 processors and 1TB of DDR4 RAM. These performance results compare the first version of threading in Julia to a mature implementation of OpenMP 3.1.

III. Results

The results of the benchmark normalized to the STINGER performance can be seen in Fig 1. The sum of runtimes across all graph scales are compared in Table IV. We see that the Julia wrapper performs within $1.5x$ of the STINGER performance in most cases. The worst case slowdown is $2.14x$. It tends to beat the C implementation for small scales, but for larger scales the C implementation comes out on top. Fig. 2 demonstrates the parallel scaling of the multithreaded BFS for a Kronecker graph of scale 27 and edgefactor 16. Fig. 2 demonstrate that Julia multithreading even in its experimental form is as effective as OpenMP parallelism for this task.

[1] https://github.com/graph500/graph500/blob/master/octave/kronecker_generator.m

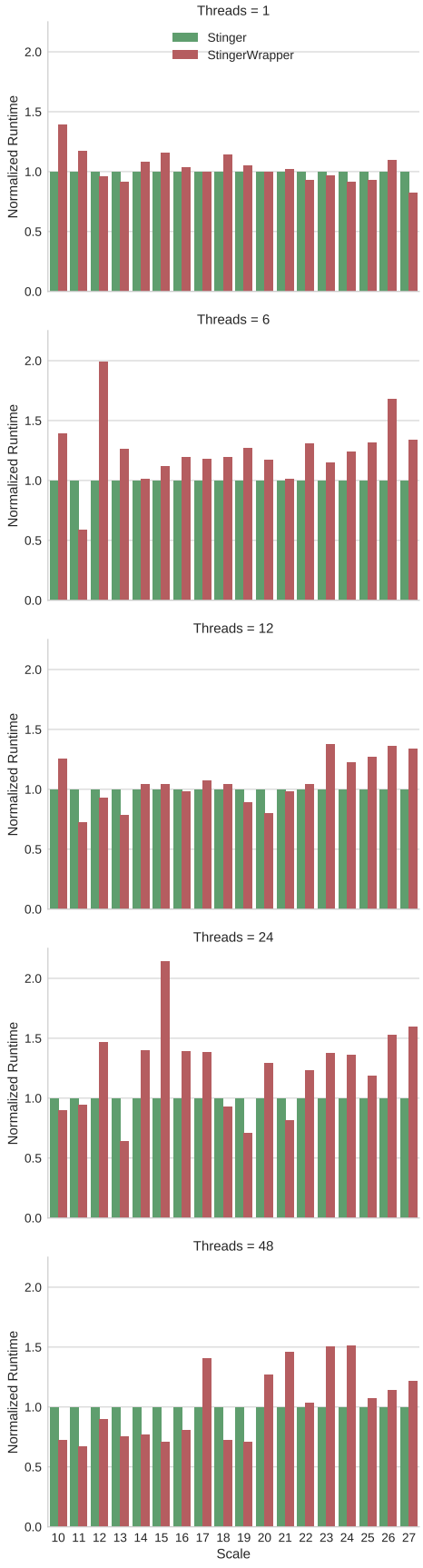

Fig. 1. Graph500 Benchmark Results (Normalized to STINGER)

TABLE IV
TOTAL TIME TO RUN GRAPH500 BFS BENCHMARK FOR ALL GRAPHS SCALE 10-27, IN MINUTES

Threads	STINGER	Stinger.jl	Slowdown
1	276.46	250.18	0.90x
6	169.93	237.21	1.40x
12	140.53	185.74	1.32x
24	97.73	145.83	1.49x
48	86.41	103.08	1.19x

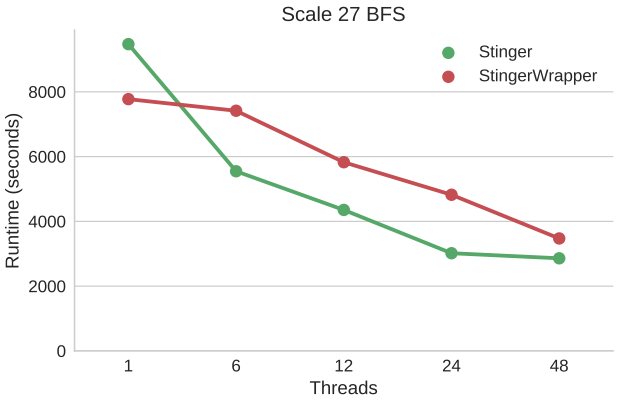

Fig. 2. Performance scaling with threads

IV. CONCLUSION

The low overhead offered by the Julia FFI and the ability to write high performance code in Julia makes it feasible to integrate low-level complex data structures in Julia without sacrificing performance. The parallelism offered by multi-threading in Julia 0.5 is mature enough to compete with OpenMP based computations using our new collections of atomics which are efficient for memory intense applications such as graph traversal workloads.

Based on our in-depth study of integrating Julia with a complex C library, we are able to draw conclusions about how to design such integrations. Memory transfer and allocation costs are driving factors in runtime. Julia is ready to support adoption of complex HPC codes using FFI to integrate low-level complex data structures.

REFERENCES

[1] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, March 2011.

[2] E. Jones, T. Oliphant, and P. Peterson, "{SciPy}: open source scientific tools for {Python}," 2014.

[3] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.

[4] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: A portable linear algebra library for high-performance computers," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1990, pp. 2–11.

[5] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "Sejits: Getting productivity and performance with selective embedded jit specialization," *Programming Models for Emerging Architectures*, vol. 1, no. 1, pp. 1–9, 2009.

[6] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, Aug. 2008, pp. 11–15.

[7] JuliaGraphs, "Lightgraphs.jl," https://github.com/JuliaGraphs/LightGraphs.jl, 2017.

[8] D. Campbell, D. Ediger, J. Poovey, and T. Goodyear, "Real-time Traffic Classification and Graph Analytics for SCinet," 2014.

[9] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos, "Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation," *Georgia Institute of Technology, Tech. Rep*, 2009.

[10] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–5.

[11] J. Leskovec and R. Sosič, "Snap: Stanford network analysis platform," 2013.

[12] J. Leskovec *et al.*, "Stanford network analysis platform," *Online: http://snap. stanford. edu/snap/index. html, f evereiro*, 2013.

[13] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal, Complex Systems*, vol. 1695, no. 5, pp. 1–9, 2006.

[14] T. P. Peixoto, "The graph-tool python library," *figshare*, 2014.

[15] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.

[16] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, "Networkit: A tool suite for large-scale complex network analysis," *arXiv preprint arXiv:1403.3005*, 2014.

[17] A. Lugowski, A. Buluç, J. R. Gilbert, and S. Reinhardt, "Scalable complex graph analysis with the knowledge discovery toolbox," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012, pp. 5345–5348.

[18] A. Buluc, E. Duriakova, A. Fox, J. R. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams, "High-productivity and high-performance analysis of filtered semantic graphs," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 237–248.

[19] D. M. Beazley *et al.*, "Swig: An easy to use tool for integrating scripting languages with c and c++." in *Tcl/Tk Workshop*, 1996.

[20] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: http://dx.doi.org/10.1137/141000671

[21] D. Beazley, "Understanding the python gil," in *PyCON Python Conference. Atlanta, Georgia*, 2010.

[22] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A Performance Evaluation of Open Source Graph Databases," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, ser. The 1st Workshop on Parallel Programming for Analytics Applications (PPAA 2014). New York, NY, USA: ACM, 2014, pp. 11–18. [Online]. Available: http://doi.acm.org/10.1145/2567634.2567638

[23] "Calling c and fortran code." [Online]. Available: https://docs.julialang.org/en/stable/manual/calling-c-and-fortran-code/

[24] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[25] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*. ACM, 1995, vol. 30, no. 8.

[26] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[27] R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald, "Frontier search," *Journal of the ACM (JACM)*, vol. 52, no. 5, pp. 715–748, 2005.

[28] E. Hein, "Dynograph," https://github.com/DynoGraph/stinger-dynograph, 2017.

[29] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, 2010.

[30] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 985–1042, 2010.

[31] J. Revels, "Benchmarktools.jl," https://github.com/JuliaCI/BenchmarkTools.jl, 2017.

[32] R. V. Thankachan, "Stingerwrapper-benchmarks," https://github.com/rohitvarkey/Stingerwrapper-Benchmarks, 2017.

APPENDIX

```
int64_t
parallel_breadth_first_search (
  struct stinger * S, int64_t nv,
  int64_t source, int64_t * marks,
  int64_t * queue, int64_t * Qhead, int64_t * level
)
{
  OMP("omp parallel for")
  for (int64_t i = 0; i < nv; i++) {
      level[i] = -1;
      marks[i] = 0;
  }

  int64_t nQ, Qnext, Qstart, Qend;
  /* initialize */
  queue[0] = source;
  level[source] = 0;
  marks[source] = 1;
  Qnext = 1;/* next open slot in the queue */
  nQ = 1;/* level we are currently processing */
  Qhead[0] = 0;/* beginning of the current frontier */
  Qhead[1] = 1;/* end of the current frontier */

  Qstart = Qhead[nQ-1];
  Qend = Qhead[nQ];

  while (Qstart != Qend) {
    OMP ("omp parallel for")
    for (int64_t j = Qstart; j < Qend; j++) {
      STINGER_FORALL_OUT_EDGES_OF_VTX_BEGIN (
        S, queue[j]
      ) {
        int64_t d = level[STINGER_EDGE_DEST];
        if (d < 0) {
          if (
          stinger_int64_fetch_add(
          &marks[STINGER_EDGE_DEST], 1
          ) == 0) {
           level[STINGER_EDGE_DEST] = nQ;
           int64_t mine = stinger_int64_fetch_add(
             &Qnext, 1
           );
           queue[mine] = STINGER_EDGE_DEST;
        }
      }
    } STINGER_FORALL_OUT_EDGES_OF_VTX_END();
  }
  Qstart = Qhead[nQ-1];
  Qend = Qnext;
  Qhead[nQ++] = Qend;
  }
  return nQ;
}
```

Listing 2: Stinger C BFS Code